IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Patent Application of

NEVILL — Atty. Ref.: 550-189

Serial No. 09/726,391 — Group: 2127

Filed: December 1, 2000 — Examiner: Bullock

For: INSTRUCTION INTERPRETATION WITHIN A DATA PROCESSING SYSTEM

Before the Board of Patent Appeals and Interferences

BRIEF FOR APPELLANT

On Appeal From Final Rejection
From Group Art Unit 2127

John R. Lastova
NIXON & VANDERHYE P.C.
8th Floor, 1100 North Glebe Road
Arlington, Virginia 22201-4714
(703) 816-4025
Attorney for Appellant
Nevill
ARM Limited

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE
BEFORE THE BOARD OF PATENT APPEALS AND INTERFERENCES

In re Patent Application of

NEVILL — Atty. Ref.: 550-189

Serial No. 09/726,391 — Group: 2127

Filed: December 1, 2000 — Examiner: Bullock

For: INSTRUCTION INTERPRETATION WITHIN A DATA PROCESSING SYSTEM

* *

April 18, 2005

Mail Stop Appeal Brief - Patents
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

APPEAL BRIEF

Sir:

I. REAL PARTY IN INTEREST

The real party in interest is the assignee, ARM Limited, a United Kingdom corporation.

II. RELATED APPEALS AND INTERFERENCES

There are no other appeals related to this subject application. There are no interferences related to this subject application.

III. STATUS OF CLAIMS

Claims 1-14 are pending. Claims 1 and 7-14 stand rejected under 35 U.S.C. §102 for anticipation based on USP 6,513,156 to Bak. Claims 2-6 stand rejected under 35 U.S.C. §103 as being unpatentable over Bak.

IV. STATUS OF AMENDMENTS

No amendment has been filed after final.

V. SUMMARY OF THE CLAIMED SUBJECT MATTER

The claims are directed to data processing systems that have an instruction interpreter that replaces a slow form instruction with a fast form instruction and that operates using a separate instruction store and data store. An example of a data processing system that operates using a separate instruction store and data store is often referred to as a Harvard architecture. An example of a separate data store and an instruction store in the form of a separate data cache and instruction cache is illustrated in Figure 1 reproduced here for convenience:

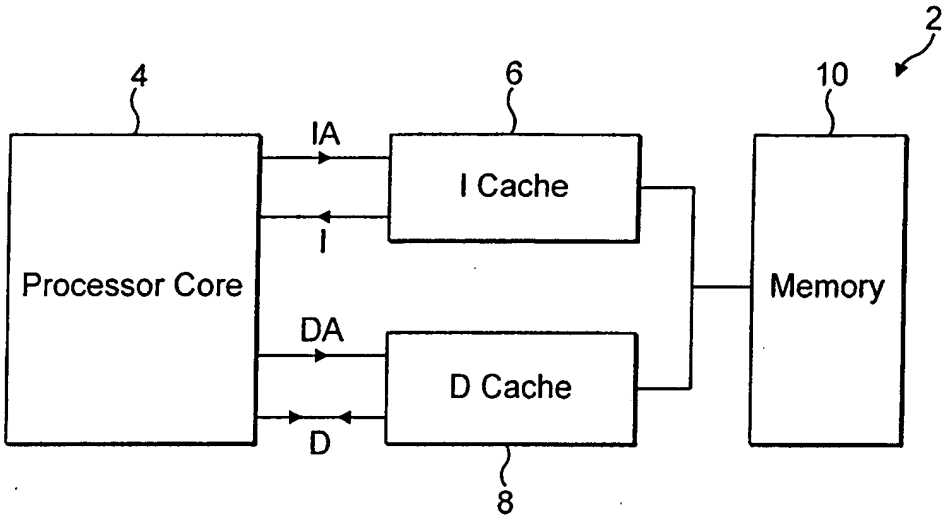


FIG. 1

In the operation of the data processing system 2, instructions to be executed are read from the main memory 10 into the instruction cache 6 and then from the instruction cache 6 into the processor core 4 where they are executed. Data words required for the data processing operation specified by the instructions or generated by those instructions are read from or written to the data cache 8.

One problem with this arrangement is how to deal with instruction code that is dynamically altered at runtime. It is known to provide an instruction interpreter that will modify a slow form of instruction to a fast form of instruction at runtime. In a Harvard architecture, the instructions are typically provided within a read only instruction store, and the writing of a modified fast form of instruction out to the data store requires a flush and reload of at least some portions of the data and instruction stores to avoid a risk of inconsistency between different forms of the same instruction being held in the instruction store and the data store. But this flush and reload reduces the system performance.

This problem is solved by having the instruction interpreter check, upon encountering a slow form instruction, whether a corresponding fast form instruction exists within the data store. If present, the interpreter replaces the slow form instruction with that fast form instruction. The slow form instruction and the fast form instruction have a common functionality when executed by the interpreter. The inventor discovered that the additional processing overhead associated with this check within the data store for a fast form of instruction is more than compensated for by the ability reliably to replace slow form instructions with fast form instructions with systems having a separate data store and instruction store. Figure 2 illustrates an example procedure for such checking (block 12) and replacement (block 14) if the corresponding fast form instruction already exists within the data store. Otherwise, a resolution procedure is performed to generate a corresponding fast form instruction (see blocks 18 and 20).

As set forth in the independent claims 1 and 14, even though the instruction interpreter can modify a slow form instruction within the instruction store to one or more possible fast form instructions and can write the fast form instruction to the data store, the instruction interpreter first checks a slow form instruction read from the instruction store for a corresponding fast form instruction already stored within the data store. If the fast form instruction is present within the data store, then the interpreter executes the fast form instruction.

An advantageous example application is one in which an unresolved memory access is dynamically replaced by a resolved memory access. The unresolved memory access typically involves a symbolic reference to the data or instructions being sought. The resolved memory access typically includes a numeric reference to this information.

A numeric reference can be directly used to return the required information, which greatly increases processing speed. Another advantageous application is a situation in which a slow form instruction invokes additional processing procedures before completion, such as calls to further processing resources, that may even be on remote systems. And as the detailed example embodiment described at page 6, line 10-page 10, line 10 and illustrated in Figures 3 and 4 demonstrates, the ability to properly replace a slow form instruction with a fast form instruction is particularly useful when interpreting Java Virtual Machine instructions.

VI. GROUNDS OF REJECTION TO BE REVIEWED ON APPEAL

The rejection to be reviewed on appeal is the anticipation rejection based on Bak.

VII. ARGUMENT

A. Bak Fails to Disclose Every Feature in Claims 1 and 14

Bak wants to increase the speed at which interpreted computer programs execute. Bak acknowledges that one way to do this is to use a just-in-time complier or to use "quick" byte codes that use an unassigned byte code to "shadow" another byte code. Bak rejects these approaches in favor of using a hybrid of virtual and native machine instructions for a "function". A portion of the bytecodes used to implement the function is selected and compiled into a corresponding sequence of native machine instructions called a "snippet". In column 7, lines 28-33, Bak states that the snippet is executed by executing a new "go-native" virtual machine instruction that replaces or overwrites the

initial virtual machine instruction of the selected function portion. The go-native bytecode references (e.g., has a pointer to) the snippet (see column 8, lines 43-44). Snippets are generated during program execution (column 9, lines 37-39) and are held in a memory space called a "snippet zone" (column 9, lines 54-55). The interpreter generates a snippet for the byte code sequence selected for replacement and overwrites the first three bytes of the sequence with the go-native bytecode and a two-byte number specifying the snippet (column 7, lines 54-62).

The Examiner equates the claimed slow-form instruction with the original virtual machine instruction and the claimed fast-form instruction with the new virtual machine instruction, i.e., the "go-native" instruction. The Examiner equates the claimed data store with the bytecode table shown in Figure 13 maintained by the system to store information about the various bytecodes (column 12, lines 53-54). The bytecode table includes "stop" snippet flags 1059 that indicate whether the bytecode should terminate snippet generation when that bytecode is encountered. Pointers 1061 to the bytecode-specific snippet code may also be stored in the bytecode table.

To establish that a claim is anticipated, the Examiner must point out where each and every limitation in the claim is found in a single prior art reference. *Scripps Clinic & Research Found. v. Genentec, Inc.*, 927 F.2d 1565 (Fed. Cir. 1991). Every limitation contained in the claims must be present in the reference, and if even one limitation is missing from the reference, then it does not anticipate the claim. *Kloster Speedsteel AB v. Crucible, Inc.*, 793 F.2d 1565 (Fed. Cir. 1986). Bak fails to satisfy this rigorous standard.

B. Bak's Intepreter Does Not Read a Slow Form Instruction to Check for a Corresponding Fast Form Instruction

Claims 1 and 14 recite features not described by Bak. For example, claim 14 recites the feature of "upon reading a slow form instruction from said instruction store, checking for a corresponding fast form instruction within said data store and, if said fast form instruction is present within said data store, then executing said fast form instruction instead of said slow form instruction." A similar recitation is found in claim 1. Contrary to the Examiner's assertion, Bak does **not** disclose that upon reading an original virtual machine instruction (slow form instruction selected for replacement), the instruction ***interpreter checks for*** a corresponding "go-native" instruction (fast form instruction) in the bytecode table (data store) and then executes the "go-native" instruction instead of the original virtual machine instruction. Rather, the "go-native" instruction simply "specifies execution of the native machine instructions so the function includes both virtual and native machine instructions." See col. 3, lines 2-5. So Bak's interpreter simply executes the "go-native" instruction and its snippet native instructions. The interpreter does not perform the claimed checking. Because the slow form instruction in Bak has already been removed (overwritten), the code which reaches Bak's interpreter already has the slow form instruction replaced with fast form instructions.

In Bak, the bytecode instructions are compiled in a compilation phase where slow form instructions are replaced with fast form instructions. The compiled and modified code is **then** subject to interpretation and execution. The Examiner cites to col. 8, lines 32-35 in Bak which state: "[w]hen the interpreter executes the go_native bytecode, the

interpreter will look up the snippet in the snippet zone specified by the go_native bytecode and then activate the native machine instructions in the snippet." It is plain from this text that the slow form instruction has already been removed and replaced with the fast form "go-native" instruction prior to interpreting and executing that instruction. So there is no need for (and indeed no teaching of) Bak's interpreter reading the slow form instruction (bytecode 2 in Bak's Figure 5 example) and then checking for a corresponding fast form instruction within a data store. Accordingly, the Examiner's contention that Bak performs the claimed checking operation simply does not square with Bak's teachings.

In the response to argument section of the official action, the Examiner switches between Bak's compiler operations and interpreter operations in an effort to defend the anticipation rejection. But this is improper. Bak clearly shows in Figure 3 that the bytecode compiler 103 is separate from the Java virtual machine interpreter 107. Figure 4 illustrates that a portion of the virtual machine instructions (slow form) input in block 201 are selected and then compiled into native machine instructions (fast form) in block 203. Column 6, line 63-column 7, line 1 explains that at step 203, "the system recognizes individual bytecodes or sequences of bytecodes that may be advantageously ***compiled***. For example, the system may generate a snippet for each Java invoke_virtual bytecode that is encountered [since] the invoke_virtual op code may be optimized when it is compiled into native machine instructions with a snippet." After Bak's ***compiler*** has compiled the selected portion of the function into a snippet of native instructions, Bak's ***interpreter*** executes the modified instructions as shown in block 303 in Figure 5. That is why Bak explains that in "order for the snippet to be executed, a new virtual machine

instruction (called "go-native" in a preferred embodiment) is executed which specifies the subsequent execution of the snippet." Col. 7, lines 28-31. The new "go-native" instruction has already replaced the old slow form instruction. See block 303 in Figure 5.

As explained above, since the slow form instruction has already been replaced by Bak's compiler with fast form instructions by Bak's compiler, Bak's interpreter does **not** then **again** read a slow form instruction and check for a corresponding fast form instruction. The interpreter simply executes the "go-native" and snippet instructions. Col. 8, lines 32-35. Claim 1 does not recite a compiler determining whether an instruction is "a new or original form instruction" as the Examiner states on page 7 of the final office action. To the contrary, claim 1 recites "said instruction ***interpreter*** is operable upon reading a slow form instruction from said instruction store to check for a corresponding fast form instruction with said data store."

C. Bak Fails to Disclose the Claimed Storage and Processing Architecture

Bak also fails to disclose the main memory, data store, and instruction store arrangement recited in claims 1 and 14. Although Bak indicates at column 5, lines 2-3 that the data processing system could include a cache, Bak does not disclose that the snippet zone/template table (which Examiner equates with the instruction store) is stored in an instruction store that is *distinct from* main memory such that it can be accessed by an instruction store port of the processor core. Bak also does not disclose that the bytecode table (which Examiner equates with the data store) is stored in an instruction store that is *distinct from* main memory such that it can be accessed by a data store port of the processor core.

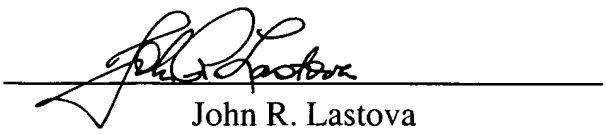
The Examiner contends that by Bak simply disclosing a "plurality of memory stores" (page 8 of the final action) that the claimed computer architecture is met. This superficial reading of the claims is improper. Claims 1 and 14 recite "a data store operable to store words from said main memory accessed by a data store port of said processor core" and "an instruction store operable to store words from said main memory accessed by an instruction store port of said processor core." Thus, the claimed arrangement, (similar to that illustrated in Figure 1 and reproduced above in the summary section), specifies that **each** of the instruction store and the data store (1) stores words from the main memory and (2) is accessed via a respective port of the processor core. Bak lacks this specific data storage and processing arrangement.

VIII. CONCLUSION

Lacking features of the claims as explained above, the Board should reverse the outstanding rejections.

Respectfully submitted,

NIXON & VANDERHYE P.C.

By: 

John R. Lastova
Reg. No. 33,149

JRL/kmm
Enclosures
Appendix A - Claims on Appeal

IX. CLAIMS APPENDIX

1. Apparatus for processing data, said apparatus comprising:

(i) a processor core;

(ii) a main memory operable to store instruction words and data words;

(iii) a data store operable to store words from said main memory accessed by a data store port of said processor core;

(iv) an instruction store operable to store words from said main memory accessed by an instruction store port of said processor core; and

(v) an instruction interpreter operable to read instruction words from said instruction store; wherein

(vi) said instruction interpreter is operable to modify a slow form instruction within said instruction store to a fast form instruction of one or more possible fast form instructions and to write said fast form instruction to said data store, said slow form instruction and said fast form instruction having a common functionality when executed by said interpreter; and

(vii) said instruction interpreter is operable upon reading a slow form instruction from said instruction store to check for a corresponding fast form instruction within said data store and, if said fast form instruction is present within said data store, then to execute said fast form instruction instead of said slow form instruction.

2. Apparatus as claimed in claim 1, wherein said instruction interpreter is a hardware based instruction translator.

3. Apparatus as claimed in claim 1, wherein said instruction interpreter is a software based interpreter.

4. Apparatus as claimed in claim 1, wherein said instruction interpreter is a combination of a hardware based instruction translator and a software based interpreter.

5. Apparatus as claimed in claim 1, wherein said data store is a data cache and said data store port is a data cache port.

6. Apparatus as claimed in claim 1, wherein said instruction store is an instruction cache and said instruction store port is an instruction cache port.

7. Apparatus as claimed in claim 1, wherein said slow form instruction results in an unresolved storage access request to one or more stored words and said fast form instruction results in a resolved storage access request to said one or more stored words.

8. Apparatus as claimed in claim 1, wherein said slow form instruction includes a symbolic reference to a required element and said fast form instruction includes a numeric reference to said required element.

9. Apparatus as claimed in claim 1, wherein said slow form instruction invokes an additional data processing procedure before completion.

10. Apparatus as claimed in claim 1, wherein said slow form instruction and said fast form instruction are Java Virtual Machine instructions.

11. Apparatus as claimed in claim 10, wherein said slow form instruction is one of:

anewarray;
checkcast;
getfield;
getstatic;
instanceof;
invokeinterface;
invokespecial;
invokestatic;
invokevirtual;
ldc;
ldc_w;
ldc2_w;

multianewarray;

new;

putfield; and

putstatic.

12. Apparatus as claimed in claim 10, wherein said fast form instruction is one of:

anewarray_quick;

checkcast_quick;

getfield_quick;

getfield_quick_w;

getfield2_quick;

getstatic_quick;

getstatic2_quick;

instanceof_quick;

invokeinterface_quick;

invokenonvirtual_quick;

invokesuper_quick;

invokestatic_quick;

invokevirtual_quick;

invokevirtual_quick_w;

invokevirtualobject_quick;

ldc_quick;

ldc_w_quick;

ldc2_w_quick;

multianewarray_quick;

new_quick;

putfield_quick;

putfield_quick_w;

putfield2_quick;

putstatic_quick; and

putstatic2_quick.

13. Apparatus as claimed in claim 10, wherein said instruction interpreter translates Java Virtual Machine instructions to native instructions of said processor core.

14. A method of processing data using an apparatus having a processor core, a main memory operable to store instruction words and data words, a data store operable to store words from said main memory accessed by a data store port of said processor core, an instruction store operable to store words from said main memory accessed by an instruction store port of said processor core, and an instruction interpreter operable to read instruction words from said instruction store; said method comprising the steps of:

(i) modifying a slow form instruction within said instruction store to a fast form instruction of one or more possible fast form instructions and to write said fast form instruction to said data store, said slow form instruction and said fast form instruction having a common functionality when executed by said interpreter; and

(ii) upon reading a slow form instruction from said instruction store, checking for a corresponding fast form instruction within said data store and, if said fast form instruction is present within said data store, then executing said fast form instruction instead of said slow form instruction.

X. EVIDENCE APPENDIX

There is no evidence appendix.

XI. RELATED PROCEEDINGS APPENDIX

There is no related proceedings appendix.